Linux Assembly Tutorial

Step-by-Step Guide

Written by: Derick Swanepoel (derick@maple.up.ac.za)
Version 1.0 - 2002-04-19, 01:50am

Download as zipfile

`JMP` Quickstart

Contents

---

1. Introduction

This tutorial is an introduction to coding assembly in Linux. There are two "versions" to accommodate various people:

- The Step-by-Step Guide: This version explains everything in detail. It assumes that you have done at least a little bit of DOS assembly, and that you have Linux on your computer (although you may not have used it much yet). Since not everyone may know how to use Linux, there are links to sections where I explain how to do basic things like use the terminal and the DOS-equivalent commands.
- The Quickstart: If you're in a hurry and just want to see a Linux assembly program, compile it and run it, this is for you. It assumes that you understand basic DOS assembly, and that you know how to use the terminal. Basically, it just points out the differences between a Linux and DOS assembly program with just enough explanation not to confuse you.

The assembler I'll be using is NASM (Netwide Assembler). Lots of the stuff in this tutorial came from other tuts and the NASM documentation – see the References section for more info.

2. Why this Tutorial?

Mainly, the reason for this tutorial is to make assembly programming easier, better and more practical by doing it in Linux instead of DOS. Also, it may teach you a bit of Linux while you're at it (unless you're already at home with it).

Programming in assembly may seem quite masochistic (and writing entire programs in it simply ridiculous), especially in these days of super-optimizing compilers and visual development tools that do just about everything for you. However, there is an advantage in understanding more about the inner workings of your processor and kernel, and assembly is a good way of learning this. Sometimes assembly can be extremely useful for sticking inline in a C/C++ program. And if your program *really* has a "need for speed", you can tweak and optimize the assembly generated by the compiler (of course, you need to be pretty elite to produce better code than today's compilers.)

Since there was this notion that we were to be taught how to use Linux during COS284 (sort of as an aside), the idea was that we would code assembly in Linux. But not Linux assembly - DOS assembly, in a DOS emulator, with a DOS text editor. Of course, this entirely defeats the purpose, but maybe it was to be done this way mainly because there aren't so many Linux assembly tutorials and sample code as for DOS. Well, here is a tutorial that'll teach you the basics of Linux assembly.

3. The Netwide Assembler (NASM)

3.1 A Note on Assemblers

Linux will almost always be intalled with the default assemblers `as` and `as86` available, and quite likely also `gas`. However, we will be using NASM, the Netwide Assembler. It uses the Intel syntax just like TASM, MASM, and other DOS assemblers, and the structure is also fairly similar. (Useless info: `as` and `gas` use the AT&T syntax, which is somewhat different – eg. all registers must be prefixed with a %, and the source operand comes before the destination. See the References for a link to a tut using `as` and AT&T syntax.)

NASM is cool because it's portable (there are Linux, Unix and DOS versions), it's free and it's powerful with lots of nice features. Trust me.

3.2 Where do I get NASM?

If you selected "Development Tools" when you installed Linux, chances are you already have NASM. It comes standard with most Linux distributions, so you don't need to download it. To check if you've got it, just ask Linux, "Where is NASM?" Here's how:

1. Open a terminal. (For some basic Linux terminal skills, go to The terminal is your friend - how to use it)
2. Type `whereis nasm` and press ENTER.

If you see a line that says something like `nasm: /usr/bin/nasm` then you're fine. If all you see is `nasm:` then you need to install NASM. Here are some instructions on how to install NASM (or anything else) on Linux.

If you feel like getting the latest and greatest version of NASM, visit their website www.cryogen.com/Nasm, or get it with FTP from our local Linux mirror ftp.kernel.za.org/pub/software/devel/nasm/binaries.

4. Introduction to Linux Assembly

4.1 Main Differences Between DOS and Linux Assembly

- In DOS assembly, most things get done with the DOS services interrupt `int 21h`, and the BIOS service interrupts like `int 10h` and `int 16h`. In Linux, all these functions are handled by the kernel. Everything gets done with "kernel system calls", and you call the kernel with `int 80h`. One of the wonderful things about Linux system calls are that there are fewer of them (about 190) than DOS, but they are far more practical (you don't have obsolete crap like functions that load casette BASIC and things left over from DOS 1.0). Linux system calls create files, handle processes and other such useful stuff - no strings attached (mmm, bad pun ;-)
- Linux is a true, 32-bit protected mode operating system, so this enables us to do real, up-to-date 32-bit assembly. This 32-bit code runs in the *flat* memory model, which basically means you don't have to worry about segments at all. This makes life a lot easier, because you never need to use a segment override or modify any segment register, and every address is 32 bits long and contains only an offset part. (If this is just a lot of waffling to you, don't worry, just know that it's good and will simplify things for you.)
- In 32-bit assembly, you use the extended 32-bit registers `EAX, EBX, ECX` and so on instead of the normal 16-bit registers `AX, BX, CX` etc.
- DOS is dead. It's 16-bit. It's obsolete. The only people that still write DOS assembly are crazy old hackers that are too attached to their 386s to throw them away. Linux assembly has practical applications (parts of the OS are written in assembler, hardware drivers are often coded in assembler).

4.2 The Parts of an Assembly Program

An assembly program can be divided into three sections:

- **The `.data` section**

  This section is for "declaring initialized data", in other words defining "variables" that already contain stuff. However this data does not change at runtime so they're not really variables. The `.data` section is used for things like filenames and buffer sizes, and you can also define constants using the `EQU` instruction. Here you can use the `DB, DW, DD, DQ` and `DT` instructions. For example:

```
section .data
        message:    db 'Hello world!'     ; Declare message to contain the bytes 'Hello world!' (witho
        msglength:  equ 12                ; Declare msglength to have the constant value 12
        buffersize: dw 1024               ; Declare buffersize to be a word containing 1024
```

- **The `.bss` section**

  This section is where you declare your variables. You use the `RESB, RESW, RESD, RESQ` and `REST` instructions to reserve uninitialized space in memory for your variables, like this:

```
section .bss
        filename:   resb    255           ; Reserve 255 bytes
        number:     resb    1             ; Reserve 1 byte
        bignum:     resw    1             ; Reserve 1 word (1 word = 2 bytes)
        realarray:  resq    10            ; Reserve an array of 10 reals
```

- **The `.text` section**

  This is where the actual assembly code is written. The `.text` section must begin with the declaration `global _start`, which just tells the kernel where the program execution begins. (It's like the main function in C or Java, only it's not a function, just a starting point.) Eg.:

```
section .text
        global _start

_start:
        pop    ebx        ; Here is the where the program actually begins
        .
        .
        .
```

As you can see, so far things are still more or less DOSish. Next we'll look at system calls in more detail, and once that is done you'll be able to write your first Linux assembly program!

4.3 Linux System Calls

Linux system calls are called in exactly the same way as DOS system calls:

1. You put the system call number in `EAX` (we're dealing with 32-bit registers here, remember)
2. You set up the arguments to the system call in `EBX, ECX`, etc.
3. You call the relevant interrupt (for DOS, `21h`; for Linux, `80h`)
4. The result is usually returned in `EAX`

There are six registers that are used for the arguments that the system call takes. The first argument goes in EBX, the second in ECX, then EDX, ESI, EDI, and finally EBP, if there are so many. If there are more than six arguments, EBX must contain the memory location where the list of arguments is stored - but don't worry about this because it's unlikely that you'll use a syscall with more than six arguments. The wonderful thing about this scheme is that Linux uses it consistently – all system calls are designed this way, there are no confusing exceptions.

Some example code always helps:

```
    mov     eax,1       ; The exit syscall number
    mov     ebx,0       ; Have an exit code of 0
    int     80h         ; Interrupt 80h, the thing that pokes the kernel and says, "Yo, do this"
```

But how do you find out what these system calls are, and what they do, and what arguments they take? Firstly, all the syscalls are listed in `/usr/include/asm/unistd.h`, together with their numbers (the value to put in `EAX` before you call `int 80h`). However, for your convenience you can simply find them in this Linux System Call Table, together with some other useful information (eg. what arguments they take). Take a look at the list of syscalls – there are things like `sys_write` (4), `sys_nice` (34) and of course `sys_exit` (1). To find out just what these things do, you can look them up in the Linux manual pages (commonly called "the manpages"). That is what the next section is about.

4.3.1 Reading the Manpages

First, open a terminal (or switch to one of the 6 consoles with `CTRL+ALT+F1` through `F6` - to get back to graphical mode press `CTRL+ALT+F7`). Say now you want to know what the "write" syscall does. Type `man 2 write` and press ENTER. This will bring up the manual page on "write" from section 2 of the manpages.

Under the **NAME** section is the syscall's name and what it does – in this case:

```
write - write to a file descriptor
```

This is the syscall you use to write to, well, a file. But you also use it to print stuff on the screen. "Why the heck is that?" you ask. See, in Linux everything is a file. Things like the screen, mice, printers, etc. are special files called "device files", but you read and write to them just like you do to a text file. This actually makes sense, because reading/writing files is one of the simplest things to do in programming, so why not do everything in the same simple way - but I digress.

Next, under the **SYNOPSIS** section you see a fairly ugly line:

```
ssize_t write(int fd, const void *buf, size_t count);
```

OK, if you know C it won't be ugly, because this is just the C definition of the syscall. As you can see, it takes three arguments: the file descriptor, followed by the buffer, and then how many bytes to write, which should be however long the buffer is. (The **DESCRIPTION** section tells us what the arguments are for.) The file descriptor (`fd`) is an integer, the buffer (`buf`) is a pointer to a memory location (that's what the * means), so it's also an integer, and the bytes to write (`count`) is of type size_t, which is also an integer. This makes sense because we put values for these arguments in the registers EBX, ECX and EDX, which are all 32-bit integers. Finally, the write syscall returns a value in EAX: the number of bytes actually written. This can be used to verify if all went well.

Now we can finally write our first Linux assembly program!

4.4 "Hello World!" in Linux Assembly

Of course, the appropriate way to begin would be to print out "Hello world!" To print to the screen, we write to the special file called STDOUT (standard output), which is file descriptor 1. Here is the program in full:

```
section .data
        hello:     db 'Hello world!',10    ; 'Hello world!' plus a linefeed character
        helloLen:  equ $-hello             ; Length of the 'Hello world!' string
                                           ; (I'll explain soon)

section .text
        global _start

_start:
        mov eax,4            ; The system call for write (sys_write)
        mov ebx,1            ; File descriptor 1 - standard output
        mov ecx,hello        ; Put the offset of hello in ecx
        mov edx,helloLen     ; helloLen is a constant, so we don't need to say
                             ;  mov edx,[helloLen] to get it's actual value
        int 80h              ; Call the kernel

        mov eax,1            ; The system call for exit (sys_exit)
        mov ebx,0            ; Exit with return code of 0 (no error)
        int 80h
```

Copy this program into a text editor of your choice (I use `vi` or SciTE), and save it as `hello.asm` in your home directory (`/home/yourname`).

4.5 Compiling and Linking

1. If you don't have a terminal or console open, open one now.
2. Make sure you are in the same directory as where you saved `hello.asm`.
3. To assemble the program, type
   `nasm -f elf hello.asm`
   If there are any errors, NASM will tell you on what line you did what wrong.
4. Now type `ld -s -o hello hello.o`
   This will link the object file NASM produced into an executable file.
5. Run your program by typing `./hello`
   (To run programs/scripts in the current directory, you must always type `./` before the name, unless the current directory is in the path.)

You should see `Hello world!` printed to the screen. Congratulations! You have just written your first assembly program in Linux!

5. More Advanced Concepts

Before I go on, you're probably wondering what that `equ $-hello` thing is doing in our Hello World program (line 3). As you may remember, when you use `equ` to declare a variable (instead of `db`, for example), you are actually declaring a constant. Declaring the length of our string as a constant is sensible because it sure isn't going to change. But how does `$-hello` turn out to be the length of 'Hello world!' ? When NASM sees a '$' it replaces it with the assembly position at the beginning of that line. (That is also the position at the end of the previous line.) So subtracting the position of a variable from '$' will give us the number of bytes between the variable and '$'. If we want to declare a variable that contains the length of a string we've declared by saying `hello: db 'Hello world!',10` then we just stick `helloLen: equ $-hello` on the next line. That will make `helloLen` equal to the number of bytes that `hello` takes up in memory, which in this case is 13 (the linefeed character also counts). Don't worry if this confuses you – just remember that it's a neat and easy way to declare the length of a string.

If you're more than just casually interested, I'd encourage you to check out the NASM documentation for more information on these things, and how to use some of the other neat features that I'm not going to mention in this tutorial.

5.1 Command Line Arguments and the Stack

Getting the command line arguments from a DOS program is not an enjoyable experience, because working with the PSP and having to worry about segments is simply a pain. In Linux things are much simpler: all arguments are available on the stack when the program starts, so to get them you just `pop` them off.

As an example, say you run a program called `program` and give it three arguments:

```
./program foo bar 42
```

The stack will then look as follows:

4	Number of arguments (argc), including the program name
program	Name of the program (argv[0])
foo	Argument 1, the first real argument (argv[1])
bar	Argument 2 (argv[2])
42	Argument 3 (argv[3]) (Note: this is the string "42", not the number 42)

Now lets write the program `program` that takes the three arguments:

```
section .text
        global _start

_start:
        pop     eax             ; Get the number of arguments
        pop     ebx             ; Get the program name
        pop     ebx             ; Get the first actual argument ("foo")
        pop     ecx             ; "bar"
        pop     edx             ; "42"

        mov     eax,1
        mov     ebx,0
        int     80h             ; Exit
```

After all that popping, EAX contains the number of arguments, EBX points to wherever "foo" is stored in memory, ECX points to "bar" and EDX to "42". This is obviously *way* more elegant and simple than in DOS. It took us just 5 lines to get the arguments and even how many there are, while in DOS it takes 14 rather complicated lines just to get *one* argument! Note that the 3rd `pop` overwrites the value we put in EBX with the 2nd `pop` (which was the program name). Unless you have a really good reason, you can usually chuck away the program name as we did here.

5.2 "Procedures" and Jumping

NB: NASM doesn't have procedure definitions like you may have used in TASM. That's because procedures don't really exist in assembly: everything is a label. So if you want to write a "procedure" in NASM, you don't use `proc` and `endp`, but instead just put a label (eg. `fileWrite:`) at the beginning of the "procedure's" code. If you want to, you can put comments at the start and end of the code just to make it look a bit more like a procedure. Here's an example in both Linux and DOS:

Linux

```
; proc fileWrite - write a string to a file
fileWrite:
   mov eax,4               ; write system call
   mov ebx,[filedesc]      ; File descriptor
   mov ecx,stuffToWrite
   mov edx,[stuffLen]
   int 80h
   ret
; endp fileWrite
```

DOS

```
proc fileWrite
   mov ah,40h              ; write DOS service
   mov bx,[filehandle]     ; File handle
   mov cl,[stuffLen]
   mov dx,offset stuffToWrite
   int 21h
   ret
endp fileWrite
```

NB[2]: I assume that you're familiar with labels and jumping to them with instructions like `JMP`, `JE` or `JGE`. Now that you've seen that "procedures" are actually labels, there is one very important thing to remember: If you are planning to return from a procedure (with the `RET` instruction), ***don't jump to it!*** As in "never!" Doing that will cause a segmentation fault on Linux (which is OK – all your program does is terminate), but in DOS it may blow up in your face with various degrees of terribleness. The rule to remember is:

> You may *jump* to *labels*, but you must call a procedure.

Calling a procedure is of course done with the `CALL` instruction. This makes life a bit difficult when you want to do things like "if-then-else". If you have a situation such as "if this happens, call procedure 1, else call procedure 2" there's only one thing to do: Jump around like a kangaroo weaving a spaghetti code masterpiece. Lets look at an example. First, here is some normal, sane code:

```
if (AX == 'w') {
   writeFile();
} else {
   doSomethingElse();
}
```

This is how you would do it in assembly:

```
        cmp     AX,'w'          ; Does AX contain 'w'?
        jne     skipWrite       ; If not, skip writing by jumping to another label, and doSomethingElse the
        call    writeFile       ; ...else call the writeFile procedure...
        jmp     outOfThisMess   ; ...and jump past all of this spaghetti

skipWrite:
        call    doSomethingElse
outOfThisMess:
        ...                     ; The rest of the program goes on here
```

Note that this is applicable to any assembly, not just Linux or NASM.

5.3 A Program with Everything

Now we can finally take a look at a program that does something remotely useful, containing almost everything we've covered. In the Quickstart version of this tutorial, I have included a Linux and a DOS version of the program we wrote in Practical 3 (the one that writes 'Hello world!' to the file given as a command line argument). Check it out and see how much simpler and logical the Linux program is compared to the DOS one.

6. Conclusion

Well, that's about it for this tutorial. I hope this has been a suitable introduction to doing assembly programming in Linux. If you have any questions, suggestions or problems, feel free to e-mail me at derick@maple.up.ac.za. This is my first tutorial and I'm no assembly hacker either, so I welcome your comments.

Good luck and happy coding!

Appendix A. The terminal is your friend - how to use it

The terminal / console is an integral and very useful part of Linux. Linux has an excellent set of command line utilities and programs, and you can control the whole system without a GUI. Sometimes this is actually easier and faster. For programming in assembly you are obviously going to have to work in the terminal, and this part will show you how.

Before you start, keep in mind that Unix/Linux is case sensitive, so "Blah" is not the same as "blah" or "blaH".

- **Opening a terminal:** If you're in KDE or Gnome, simply click on the terminal icon or browse the "Start"-menu for a terminal program. Alternatively you can switch to one of the 6 text-mode consoles by pressing `CTRL+ALT+F1` through `F6`. To get back to graphics mode press `CTRL+ALT+F7`. When you open a terminal or log in on a console, you are plonked into your home directory, which is called whatever your username is. You are presented with a prompt that looks something like this:

```
[delta@quantumcow asmtut]$
```

  The part before the '@' tells you your username (mine is **delta**), then the computer name (**quantumcow**), and then the top-level current directory (**asmtut**).
- **Finding out where you are:** At the prompt, type `pwd`. This will show you the "present working directory", in other words where you are now. For example:

```
[delta@quantumcow asmtut]$ pwd
/home/delta/asmtut
```

- **Changing directories:** To change to another directory you use the `cd` command. Note that it works a bit different than the DOS `cd`. Firstly, there must be a space between "cd" and the directory name. Secondly, in Unix/Linux, the directory separator is a forward slash (/) not a backslash (\). To change to the parent directory of the current one, you go `cd ..`. To go up two levels in the directory tree, type `cd ../..`. Here are some examples – try them out:

```
[delta@quantumcow asmtut]$ cd /usr/share/doc
[delta@quantumcow doc]$ pwd
/usr/share/doc
[delta@quantumcow doc]$ cd ..
[delta@quantumcow share]$ cd ../..
[delta@quantumcow /]$
```

  At the end of this example, you end up in the root directory, `/` (similar to C:\). Now to get back to your home directory, type `cd ~` The tilde (~) is a shortcut for your home directory.
- **Finding out what's in a directory:** To list the current directory's contents, type `ls`. If you're in your home directory and haven't used Linux much, there probably won't be many files. Change to a directory like `/etc` and list its contents – lots of files! (The /etc directory is where most of the system's configuration files are stored.) To get some more info, you can do a "long list" by typing `ls -l` or simply `ll`. In my home directory it looks like this:

```
[delta@quantumcow delta]$ ls
All.m3u  autosave  Desktop  GNUstep    phy351dist.sh  Playlist.m3u     tcl
asmtut   colors.sh  foo      image.tgz  playlist       Singularity.jpg  TclTutor
[delta@quantumcow delta]$ ll
total 236
-rw-rw-r--    1 delta    delta        4362 Apr 17 20:08 All.m3u
drwxr-xr-x    2 delta    delta        4096 Apr 18 23:21 asmtut
drwxrwxr-x    2 delta    delta        4096 Apr 18 23:12 autosave
-rwxr-xr-x    1 root     root           96 Apr 15 21:45 colors.sh
drwx------    3 delta    delta        4096 Apr 10 21:58 Desktop
drwxrwxr-x    2 delta    delta        4096 Apr  9 19:43 foo
drwxrwxr-x    5 delta    delta        4096 Apr 10 21:58 GNUstep
-rw-rw-r--    1 delta    delta       89758 Apr 19 00:12 image.tgz
-rwxr-xr-x    1 delta    delta         753 Apr  9 21:50 phy351dist.sh
-rw-r--r--    1 root     root         1942 Apr 15 21:46 playlist
-rw-rw-r--    1 delta    delta        2146 Apr 17 20:08 Playlist.m3u
-rw-r--r--    1 delta    delta       89707 Apr 19 00:11 Singularity.jpg
drwxr-xr-x    3 delta    delta        4096 Apr 17 19:43 tcl
drwxr-xr-x    2 delta    delta        4096 Apr 10 23:10 TclTutor
[delta@quantumcow delta]$
```

  Directories are highlighted in blue, executable files in green, compressed archives in red and images in purple.
- **Finding out what's in a file:** You can use either `cat` or `less` to view the contents of a file. Use `less` when you know the output will be more than one screen long, because `less` will pause at every screen of text, and allows you to scroll up and down.

```
[delta@quantumcow asmtut]$ cat foo.txt
Hello, world!
```

- **The wonders of tab-completion:** Linux has a wonderful feature called tab-completion. Almost anywhere, when you are typing in the name of a file or directory, you can press the `TAB` key and Linux will complete the name for you. This is especially useful when you're dealing with things that have long names. Try find a directory with really long directory names in it, and then `cd` to one of them using tab-completion. For example, type `cd /usr/share/doc/cons` and press `TAB`. Tada! Linux has completed the name for you, (`console-tools-19990829/`) and you can just press `ENTER`.

  If you try typing `cd /usr/share/doc/proc` and then press `TAB` you'll hear a beep. That's because there are more than one directory in /usr/share/doc that starts with "proc", so Linux doesn't know which one you want. If you press `TAB` again it will display the

directories starting with "proc". Now you can type some more letters of the directory you want (just enough to identify it uniquely will do), and press `TAB` again. Neat, isn't it?

It is often useful to have more than just one terminal open, for example one to compile your program in and another to read manpages with. Also, most window managers (KDE, Gnome, IceWM are window managers) allow you to work on multiple desktops. So when one desktop gets cluttered with windows, just go to another one and "start with a clean slate!" In KDE and Gnome you'll see four numbered little squares on the taskbar. Click on one and it takes you to that desktop. (You can have up to 10 desktops if you want.) I have on one rare occasion worked on 4 consoles and 3 desktops at the same time because of all the different stuff I was doing!

Appendix B. Installing NASM (and other stuff) on Linux

In order to install programs on your Linux system, you must be root (administrator). You can decide whether you want to do this with the GUI utilities or in a terminal – I recommend you try both, for the added experience ;)

Using KDE / Gnome

If you're working in KDE / Gnome, installing things is fairly straightforward:

1. If you are logged in as a normal user, log out and log in as root.
2. Pop your Linux CD in your CD-ROM drive, and it should be automounted (NASM is on the **2nd** CD). You can then use a file manager (Konqueror, Nautilus) to go to `/mnt/cdrom` where you should see the contents of the Linux CD. If there's nothing, go to the desktop, right-click on the CD-ROM icon and click "Mount".
3. In `/mnt/cdrom`, go to `RedHat`, then `RPMS`. You should see a lot of files with the extension `.RPM` – look for `nasm-0.98-8.i386.rpm`.
4. Click/double-click on it to open it with the package manager. From there it shouldn't be too much of a mission to install. Also install `nasm-doc-0.98-8.i386.rpm`, the documentation for NASM.

Using the Terminal

Installing stuff by means of a terminal isn't difficult either:

1. If you weren't logged in as root, become root in the terminal by typing `su` (for "substitute user"). After entering your password, you will now be logged in on that terminal as root.
2. Mount the CD-ROM by typing `mount /dev/cdrom`
3. Change to the packages directory: `cd /mnt/cdrom/RedHat/RPMS`
4. Install NASM with the RedHat Package Manager, by typing: `rpm -i nasm-0.98-8.i386.rpm`
   (Hint: use tab-completion!)
   Also install `nasm-doc-0.98-8.i386.rpm`, the documentation for NASM.
5. If you su'd to become root, type `exit` to log out and stop being root.

If everything was successful, congratulations! You're well on your way to becoming an elite Linux user. If something broke, feel free to e-mail me and I'll do my best to help. Good luck, and happy hacking in Linux!

Appendix C. References

Writing a useful program with NASM
The NASM documentation
Introduction to UNIX assembly programming
Linux Assembler Tutorial by Robin Miyagi
Section 2 of the manpages